Exploratory Data Analysis with GGplot

Russell Almond

August 27, 2020

Tidyverse Software

For this example, we are going to use GGplot, which is part of the tidyverse. Tidyverse is an extra layer on top of R which makes it easy to manipulate data as a kind of a workflow. Note that tidyverse is actually a meta-package: it downloads a number of generally useful packages, including GGplot (GG stands for *Grammar of Graphics*, a book about how to build up complex plots from smaller pieces.)

The command `install.packages()` installs packages, that is, it downloads them from the CRAN library to your local computer. The command `library()` tells R that you want to use that package in this session. You need to run `library()` every time, but you only need to run `install.packages()` once.

```
if (!("tidyverse" %in% row.names(installed.packages()))) {
  install.packages("tidyverse",repos="https://cloud.r-project.org",dependencies=TRUE)
}
library(tidyverse)
```

```
## -- Attaching packages ------------------------------------------------------------------ tidyverse 1.3

## v ggplot2 3.2.1     v purrr   0.3.3
## v tibble  2.1.3     v dplyr   0.8.4
## v tidyr   1.0.2     v stringr 1.4.0
## v readr   1.3.1     v forcats 0.4.0

## -- Conflicts ------------------------------------------------------------------ tidyverse_conflicts
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

Dplyr tools

Tools for manipulating data.

Tibbles

For this exercise we will use the data set `state.x77` which comes with R. You can find more information about this data set by doing:

```
help(state.x77)
```

A `tibble` is a data structure with rows corresponding to cases and columns to variables. It is a *tidy* version of a data frame.

```
as_tibble(state.x77) %>% add_column(region=state.region,name=state.name,code=state.abb,center_x=state.c
View(state77)
state77
```

```
## # A tibble: 50 x 13
```

```
##    Population Income Illiteracy `Life Exp` Murder `HS Grad` Frost   Area region
##         <dbl>  <dbl>      <dbl>      <dbl>  <dbl>     <dbl> <dbl>  <dbl> <fct>
##  1       3615   3624        2.1       69.0   15.1      41.3    20  50708 South
##  2        365   6315        1.5       69.3   11.3      66.7   152 566432 West
##  3       2212   4530        1.8       70.6    7.8      58.1    15 113417 West
##  4       2110   3378        1.9       70.7   10.1      39.9    65  51945 South
##  5      21198   5114        1.1       71.7   10.3      62.6    20 156361 West
##  6       2541   4884        0.7       72.1    6.8      63.9   166 103766 West
##  7       3100   5348        1.1       72.5    3.1      56     139   4862 North~
##  8        579   4809        0.9       70.1    6.2      54.6   103   1982 South
##  9       8277   4815        1.3       70.7   10.7      52.6    11  54090 South
## 10       4931   4091        2         68.5   13.9      40.6    60  58073 South
## # ... with 40 more rows, and 4 more variables: name <chr>, code <chr>,
## #   center_x <dbl>, center_y <dbl>
```

The `View()` command opens the data frame/matrix/tibble in another window.

- Try `state77` in the console. The tibble is slightly different from the data frame in the way it prints.
- Tibble and data frames are pretty much interchangeable. (Where they aren't use `as.data.frame()` or `as_tibble()` to convert.

Note the type of the variables are shown in the display of the tibble. The name and postal code are left as strings, but region is a factor (with four levels). In a data frame, the string variables are automatically converted to factors, which is not always what you want.

- Use `read_csv()` instead of `read.csv()` to load a CSV file as a tibble instead of a data frame.

The Pipe

The special operator `%>%` can be used to chain operations together.

The expression above gives an example. The output of `as_tibble()` is passed to the `add_column()` which is then passed to the assignment operator `->`.

Note the backward arrow `->`. This is like the usual assignment operator `<-` except now the name of the variable is on the right instead of the left.

A typical chain looks like:

data `%>%` `select`(*variables*) `%>%` `filter`(*cases*) `%>%` *analysis*() `->` *result*

Or maybe the *analysis* is replaced with a call to *ggplot* to make a plot.

Selecting Variables

The `select()` command can be used to select a subset of variables.

```r
state77 %>% select(code,Population,Income)
```

```
## # A tibble: 50 x 3
##    code  Population Income
##    <chr>      <dbl>  <dbl>
##  1 AL          3615   3624
##  2 AK           365   6315
##  3 AZ          2212   4530
##  4 AR          2110   3378
##  5 CA         21198   5114
##  6 CO          2541   4884
##  7 CT          3100   5348
##  8 DE           579   4809
##  9 FL          8277   4815
## 10 GA          4931   4091
## # ... with 40 more rows
```

```r
state77 %>% select(code,region:code)
```

```
## # A tibble: 50 x 3
##    code  region    name
##    <chr> <fct>     <chr>
##  1 AL    South     Alabama
##  2 AK    West      Alaska
##  3 AZ    West      Arizona
##  4 AR    South     Arkansas
##  5 CA    West      California
##  6 CO    West      Colorado
##  7 CT    Northeast Connecticut
##  8 DE    South     Delaware
##  9 FL    South     Florida
## 10 GA    South     Georgia
## # ... with 40 more rows
```

```r
state77 %>% select(-name)
```

```
## # A tibble: 50 x 12
##    Population Income Illiteracy `Life Exp` Murder `HS Grad` Frost   Area region
##         <dbl>  <dbl>      <dbl>      <dbl>  <dbl>     <dbl> <dbl>  <dbl> <fct>
##  1       3615   3624        2.1       69.0   15.1      41.3    20  50708 South
##  2        365   6315        1.5       69.3   11.3      66.7   152 566432 West
##  3       2212   4530        1.8       70.6    7.8      58.1    15 113417 West
##  4       2110   3378        1.9       70.7   10.1      39.9    65  51945 South
##  5      21198   5114        1.1       71.7   10.3      62.6    20 156361 West
##  6       2541   4884        0.7       72.1    6.8      63.9   166 103766 West
##  7       3100   5348        1.1       72.5    3.1      56     139   4862 North~
##  8        579   4809        0.9       70.1    6.2      54.6   103   1982 South
##  9       8277   4815        1.3       70.7   10.7      52.6    11  54090 South
## 10       4931   4091        2         68.5   13.9      40.6    60  58073 South
## # ... with 40 more rows, and 3 more variables: code <chr>, center_x <dbl>,
## #   center_y <dbl>
```

```r
state77 %>% select(code,starts_with("center"))
```

```
## # A tibble: 50 x 3
##    code  center_x center_y
```

```
##    <chr>     <dbl>    <dbl>
##  1 AL        -86.8     32.6
##  2 AK       -127.      49.2
##  3 AZ       -112.      34.2
##  4 AR        -92.3     34.7
##  5 CA       -120.      36.5
##  6 CO       -106.      38.7
##  7 CT        -72.4     41.6
##  8 DE        -75.0     38.7
##  9 FL        -81.7     27.9
## 10 GA        -83.4     32.3
## # ... with 40 more rows
```

Usually having more columns than you need is harmless.

For example, using `lm()` to fit a regression of `ggplot()` to make a plot will just use the variables referenced in the model or plot description.

However, sometimes is it easier to work with a smaller subset of the data with just the stuff you need.

Making New Variables

We already saw the `add_column()` function for adding columns.

The `mutate()` function adds new columns as a function of the old ones:

```
state77 %>% mutate(Pop_Density=Population/Area) -> state77a
state77a
```

```
## # A tibble: 50 x 14
##    Population Income Illiteracy `Life Exp` Murder `HS Grad` Frost   Area region
##         <dbl>  <dbl>      <dbl>      <dbl>  <dbl>     <dbl> <dbl>  <dbl> <fct>
##  1       3615   3624        2.1       69.0   15.1      41.3    20  50708 South
##  2        365   6315        1.5       69.3   11.3      66.7   152 566432 West
##  3       2212   4530        1.8       70.6    7.8      58.1    15 113417 West
##  4       2110   3378        1.9       70.7   10.1      39.9    65  51945 South
##  5      21198   5114        1.1       71.7   10.3      62.6    20 156361 West
##  6       2541   4884        0.7       72.1    6.8      63.9   166 103766 West
##  7       3100   5348        1.1       72.5    3.1      56     139   4862 North~
##  8        579   4809        0.9       70.1    6.2      54.6   103   1982 South
##  9       8277   4815        1.3       70.7   10.7      52.6    11  54090 South
## 10       4931   4091        2         68.5   13.9      40.6    60  58073 South
## # ... with 40 more rows, and 5 more variables: name <chr>, code <chr>,
## #   center_x <dbl>, center_y <dbl>, Pop_Density <dbl>
```

Recoding Variables

Recoding is important because sometimes the way the variable is stored in the data file is not the same as the way we want to analyze it.

- Factor variables can represent categories with integer values or string labels.
  - Often there is a *code book* which maps integer category labels to string values. For example:

1. Female
2. Male

The `factor()` function creates factor variables.

```r
factor(c(1,1,1,2,2,2),levels=1:2,labels=c("Female","Male"))
```

```
## [1] Female Female Female Male   Male   Male
## Levels: Female Male
```

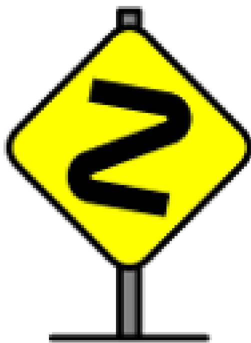
```r
factor(c("Male","Male","Male","Female","Female","Female"),levels=c("Male","Female"))
```

```
## [1] Male   Male   Male   Female Female Female
## Levels: Male Female
```

```r
ordered(c("H","H","M","M","L","L"), levels=c("L","M","H"))
```

```
## [1] H H M M L L
## Levels: L < M < H
```

- The `levels` argument tells R how the data are coded (in the case of integer coding).
- The `labels` argument gives the names for the levels (if omitted it is the same as `levels`).



The `ordered()` function produces an ordered variable as opposed to `factor()` which produces a nominal one. This only makes a difference in a few places. Probably the most important one is how they are used in an Analysis of Variance (ANOVA). That is covered in EDF 5402.

Note Bene! The `read_csv()` function which is part of the tidyverse will read factor variables as either character or integer variables, depending on how they are coded. So you will need to use `mutate(x=factor(x))` to convert `x` into a factor.

The function `parse_factor()` is almost the same, but gives a warning if some of the levels aren't recognized.

```r
factor(c("Male","Female","Non-binary"),levels=c("Male","Female"))
```

```
## [1] Male   Female <NA>
## Levels: Male Female
```

```r
parse_factor(c("Male","Female","Non-binary"),levels=c("Male","Female"))
```

```
## Warning: 1 parsing failure.
## row col           expected     actual
##   3  -- value in level set Non-binary
```

```
## [1] Male   Female <NA>
## attr(,"problems")
## # A tibble: 1 x 4
##     row   col expected           actual
##   <int> <int> <chr>              <chr>
## 1     3    NA value in level set Non-binary
## Levels: Male Female
```

Another way to do the coding is to use * `recode()` (makes a character or numeric value) * `recode_factor()` (makes a factor variable)

The first argument is the vector to be recorded, the remaining arguments are the values to be replaced.

```
recode_factor(c(1,1,1,2,2,2),`1`="Male",`2`="Female")
```

```
## [1] Male   Male   Male   Female Female Female
## Levels: Male Female
```

```
recode_factor(c(1,1,1,2,2,2),"Male","Female")
```

```
## [1] Male   Male   Male   Female Female Female
## Levels: Male Female
```

```
recode_factor(c("M","M","F","F"),M="Male",F="Female")
```

```
## [1] Male   Male   Female Female
## Levels: Male Female
```

```
recode_factor(c("White","Black","Latinx","Other"),White="White",.default="Non-White")
```

```
## [1] White     Non-White Non-White Non-White
## Levels: White Non-White
```

Note how we used the last version to collapse several categories into one. This is often useful, particularly when the number of subjects in one category is small.

Recoding NAs

A special case of recoding comes about with missing values.

In R, these are called `NA` (for Not Applicable).

- `NA`s are contagious: `NA` + anything is still `NA`.

```
NA+5
```

```
## [1] NA
```

```
mean(c(1,2,NA))
```

```
## [1] NA
```

```
mean(c(1,2,NA),na.rm=TRUE)
```

```
## [1] 1.5
```

- `NaN` (not a number) is similar but it comes from nonsense arthimatic (taking log of negative number).
- `NA`s can be coded in many different ways in a data set:
  - Leave the value blank.
  - Special character, e.g., `.` or `*`
  - Special String, e.g., `NA`
  - Nonsense numeric value, e.g., `-9`

When using nonsense numeric values, it is important to pick a value that is not plausible, e.g., a large negative value. That way, if you accidently forget to convert, you can know that something is wrong.

The function `na_if()` can be used to replace a value with NAs.

```
na_if(c(1:5,-9),-9)
```

```
## [1]  1  2  3  4  5 NA
```

```r
starwars %>% select(name,eye_color) %>%
  mutate(eye_color=na_if(eye_color,"unknown"))
```

```
## # A tibble: 87 x 2
##    name               eye_color
##    <chr>              <chr>
##  1 Luke Skywalker     blue
##  2 C-3PO              yellow
##  3 R2-D2              red
##  4 Darth Vader        yellow
##  5 Leia Organa        brown
##  6 Owen Lars          blue
##  7 Beru Whitesun lars blue
##  8 R5-D4              red
##  9 Biggs Darklighter  brown
## 10 Obi-Wan Kenobi     blue-gray
## # ... with 77 more rows
```

The function `replace_na()` goes in the opposite direction.

For example, we might want to treat missing values as score of 0 on a test.

```r
replace_na(c(1,1,0,0,NA),0)
```

```
## [1] 1 1 0 0 0
```

Logical Tests

The function `if_else()` is also useful for splitting data sets up into groups.

We can see the form in:

```r
args(if_else)
```

```
## function (condition, true, false, missing = NULL)
## NULL
```

Note that condition is a logical expression which should yeild a true or false value for every row of the tibble. The variable `true` is the value to use if true, `false` the value to use if false, and `missing` the value to use if missing.

```r
int5 <- -5:5
if_else(int5<0,"-","+")
```

```
##  [1] "-" "-" "-" "-" "-" "+" "+" "+" "+" "+" "+"
```

```r
if_else(int5<0,-int5,int5) #Absolute value
```

```
##  [1] 5 4 3 2 1 0 1 2 3 4 5
```

```r
na_if(int5,0)
```

```
##  [1] -5 -4 -3 -2 -1 NA  1  2  3  4  5
```

```r
if_else(na_if(int5,0)<0 ,"-","+","0")
```

```
##  [1] "-" "-" "-" "-" "-" "0" "+" "+" "+" "+" "+"
```

Here are the common logical tests:

- `==` – equals (don't confuse this with `=` assignment.)

- `!=` – not equals
- `<`, `<=`, `=>`, `>` – less than, &c.
- `!` – Not (true if the rest of the expression is false)
- `is.na()` – True if the value is NA, false otherwise. (Also, `!is.na()`)
- `&` – logical and (true when LHS and RHS are true)
- `|` – logical or (true if either LHS or RHS is true)
- `%in%` – True if value is in list.

```
drupes <- c("Almond","Cashew","Walnut")
c("Peanut","Almond","Hazelnut","Macademia","Cashew") %in% drupes
```

```
## [1] FALSE  TRUE FALSE FALSE  TRUE
```

Selecting Cases

Very often instead of setting the value to NA, we just want to exclude that row from the data set.

The command `filter()` does this.

```
state77 %>% filter(!(code %in% c("AK","HI")))
```

```
## # A tibble: 48 x 13
##    Population Income Illiteracy `Life Exp` Murder `HS Grad` Frost   Area region
##         <dbl>  <dbl>      <dbl>      <dbl>  <dbl>     <dbl> <dbl>  <dbl> <fct>
##  1       3615   3624        2.1       69.0   15.1      41.3    20  50708 South
##  2       2212   4530        1.8       70.6    7.8      58.1    15 113417 West
##  3       2110   3378        1.9       70.7   10.1      39.9    65  51945 South
##  4      21198   5114        1.1       71.7   10.3      62.6    20 156361 West
##  5       2541   4884        0.7       72.1    6.8      63.9   166 103766 West
##  6       3100   5348        1.1       72.5    3.1      56     139   4862 North~
##  7        579   4809        0.9       70.1    6.2      54.6   103   1982 South
##  8       8277   4815        1.3       70.7   10.7      52.6    11  54090 South
##  9       4931   4091        2         68.5   13.9      40.6    60  58073 South
## 10        813   4119        0.6       71.9    5.3      59.5   126  82677 West
## # ... with 38 more rows, and 4 more variables: name <chr>, code <chr>,
## #   center_x <dbl>, center_y <dbl>
```

Sometimes we want to temporarily remove the biggest values or the smallest values so we can see the details in a plot.

```
state77 %>% select(name,Area) %>% filter(Area <200000)
```

```
## # A tibble: 48 x 2
##    name          Area
##    <chr>        <dbl>
##  1 Alabama      50708
##  2 Arizona     113417
##  3 Arkansas     51945
##  4 California  156361
##  5 Colorado    103766
##  6 Connecticut   4862
##  7 Delaware      1982
##  8 Florida      54090
##  9 Georgia      58073
```

```
## 10 Hawaii        6425
## # ... with 38 more rows
```

Sometimes we want to create subsets of the data that just have fewer cases.

The functions `sample_frac()` and `sample_n()` specify the size of the sample in fraction of the original data or absolute size.

The function `slice()` will select a contiguous range of cases, which is useful when looping through the data.

Calculating Summary Statistics

Pipe the output of the select and filter command into `summarize()`:

```
state77 %>% summarize(N=n(),Income=mean(Income),Population=mean(Population))
```

```
## # A tibble: 1 x 3
##       N Income Population
##   <int>  <dbl>      <dbl>
## 1    50  4436.      4246.
```

Here are some useful functions to use with `summarize()`:

- `n()`, `n_distinct()`, `sum(!is.na())` – Count, count of unique values, count of non-missing values.
- `mean()`, `median()` – Measures of center
- `min()`, `max()`, `quantile()` – Position other than the center.

```
state77 %>% select(Population) %>% summarize(Min=min(Population),Q1=quantile(Population,.25),Q2=median(P
```

```
## # A tibble: 1 x 5
##     Min    Q1    Q2    Q3   Max
##   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1   365 1080. 2838. 4968. 21198
```

- `sd()`, `IQR()`, `mad()` – measures of scale.
- `sum()`, `prod()` – Arithmetic
- `sum()`, `any()`, `all()` – Summarize logical expressions (count number true, true if all are true, true if any is true).

All of these functions have an optional argument `na.rm`. If there are NAs, you usually want to include `na.rm=TRUE`, as otherwise the value will be NA.

Summarizing Multiple columns.

Often, you want to do the same summary on several columns.

The function `summarize_all()` does that.

```
state77 %>% select(Area,Population) %>% summarize_all(mean,na.rm=TRUE)
```

```
## # A tibble: 1 x 2
##     Area Population
##    <dbl>      <dbl>
## 1 70736.      4246.
```

You can use multiple statsitics by putting them in a list.

```
state77 %>% select(Area,Population) %>% summarize_all(list(mean=mean,sd=sd))
```

```
## # A tibble: 1 x 4
##   Area_mean Population_mean Area_sd Population_sd
##       <dbl>           <dbl>   <dbl>         <dbl>
```

```
## 1    70736.            4246.  85327.          4464.
```

The function `summarize_at()` combines the `select()` and `sumarize()`.

The function `summarize_if()` allows the selection of columns based on logical criteria.

Calculating Statistics by Group

Very often we want to be to compare groups. We can use the function `group_by()` to split the data set by a factor variable.

```
state77 %>% group_by(region) %>% select(Area,Population) %>% summarize_all(list(mean=mean,sd=sd))
```

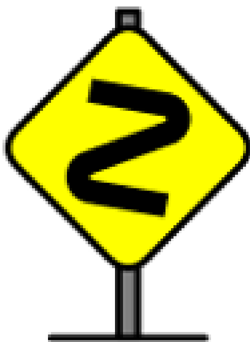
```
## Adding missing grouping variables: `region`
```

```
## # A tibble: 4 x 5
##   region        Area_mean Population_mean Area_sd Population_sd
##   <fct>             <dbl>           <dbl>   <dbl>         <dbl>
## 1 Northeast        18141            5495.  18076.         6080.
## 2 South            54605.           4208.  57965.         2780.
## 3 North Central    62652            4803   14967.         3703.
## 4 West            134463            2915. 134982.         5579.
```

```
state77 %>% group_by(region) %>%
  select(Area,Population) %>%
  summarise_all(list(Min=min,Q1=function(x){quantile(x,.25)},Q2=median,Q3=function(x){quantile(x,.75)},
```

```
## Adding missing grouping variables: `region`
```

```
## # A tibble: 4 x 11
##   region Area_Min Population_Min Area_Q1 Population_Q1 Area_Q2 Population_Q2
##   <fct>     <dbl>          <dbl>   <dbl>         <dbl>   <dbl>         <dbl>
## 1 North~     1049            472    7521           931     9027          3100
## 2 South      1982            579   37294.         2622.   46113          3710.
## 3 North~    36097            637   55427          2096    62906          4255
## 4 West       6425            365   82677           746   103766          1144
## # ... with 4 more variables: Area_Q3 <dbl>, Population_Q3 <dbl>,
## #   Area_Max <dbl>, Population_Max <dbl>
```



The `function(){}` makes an anonymous function. This gets around the problem that `quantile()` needs two arguments, but `summarize_all()` expects a function of just one.

The cheat sheet.

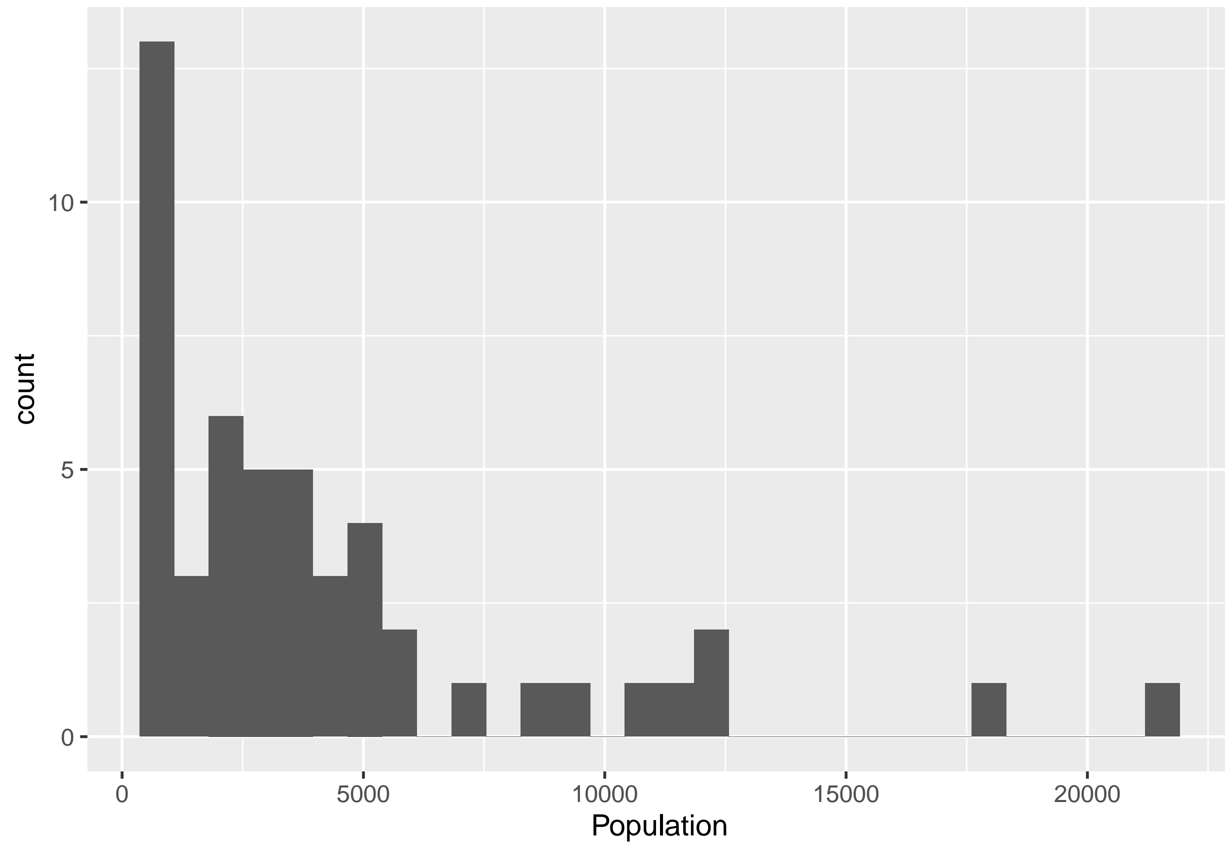
You can find a handy list of dplyr and other tidyverse commands for manipulating data by selected "Help > Cheat Sheets > Data Mainpulation with dplyr" from the RStudio menu.

Graphics

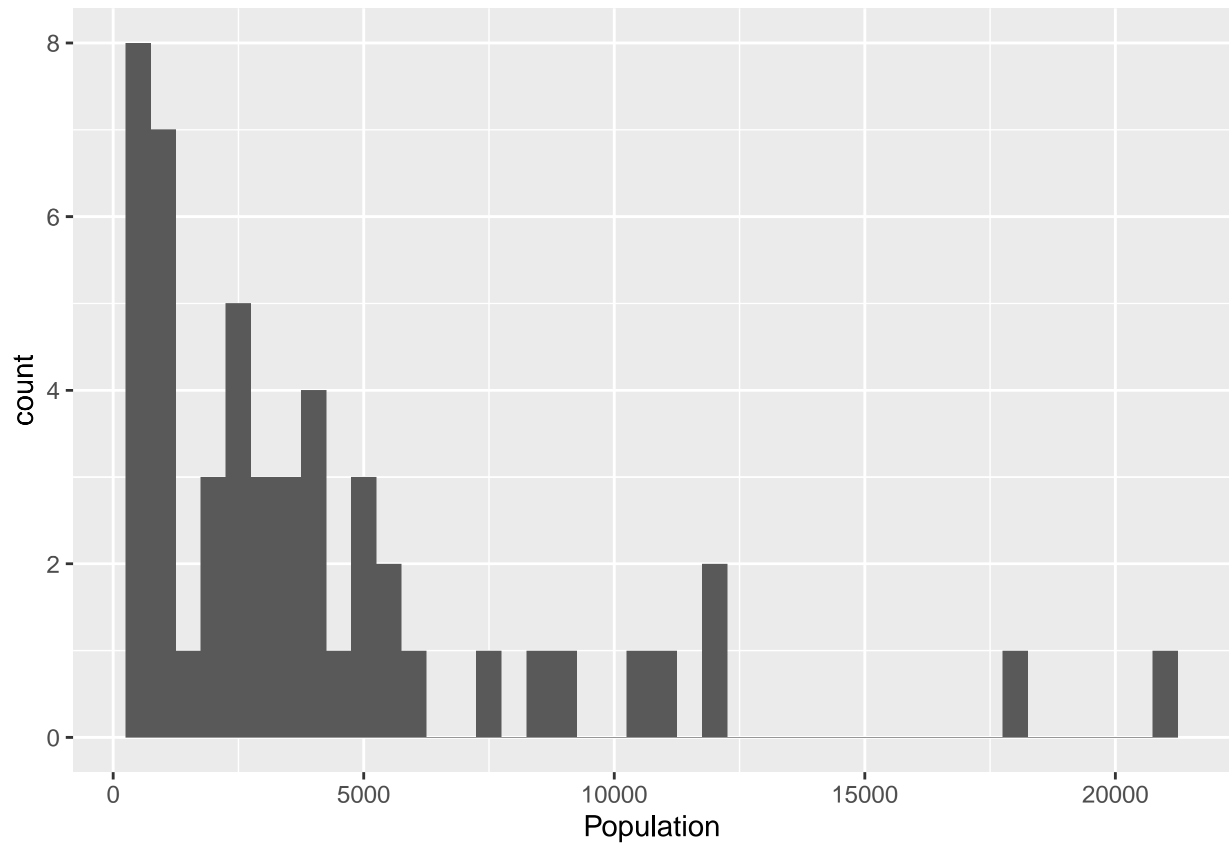
Making Histograms

```
ggplot(state77,aes(Population)) + geom_histogram()
```

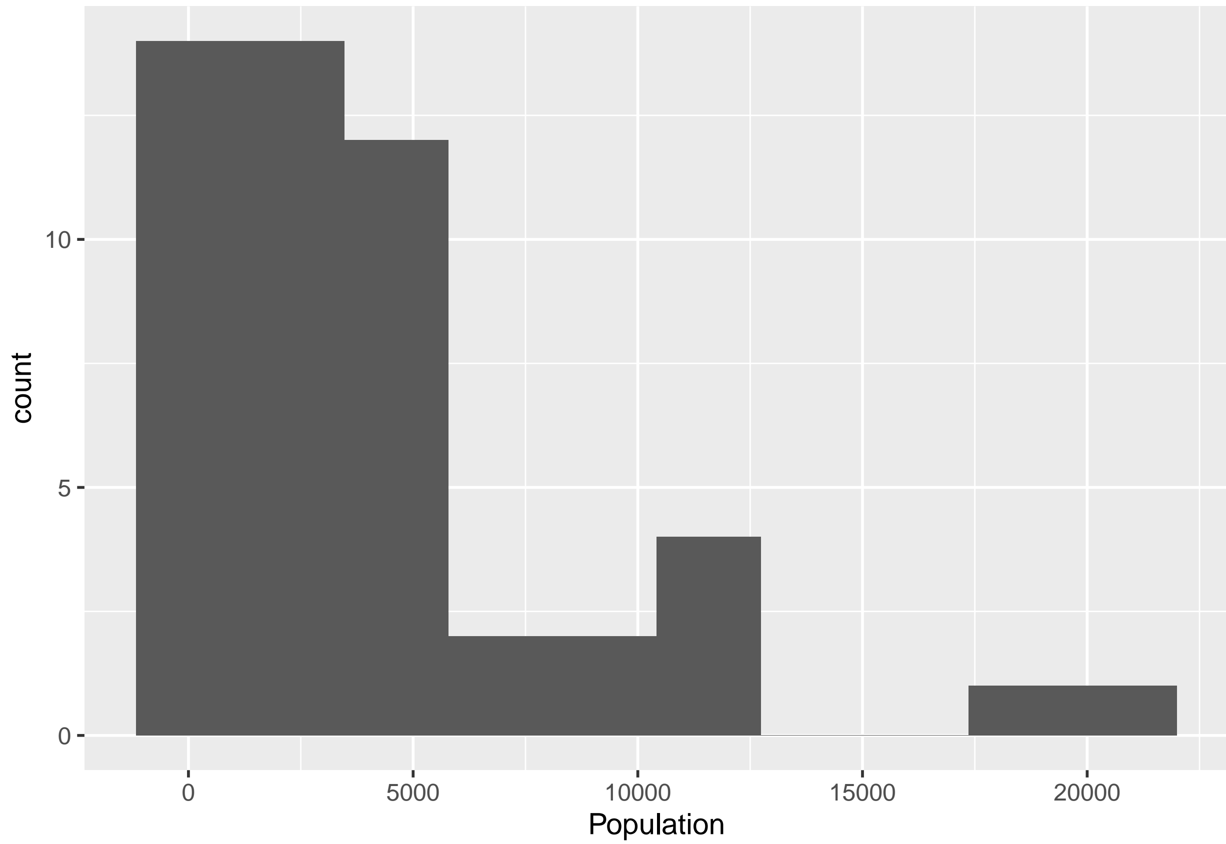
```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```




```
ggplot(state77,aes(Population)) + geom_histogram(binwidth=500)
```

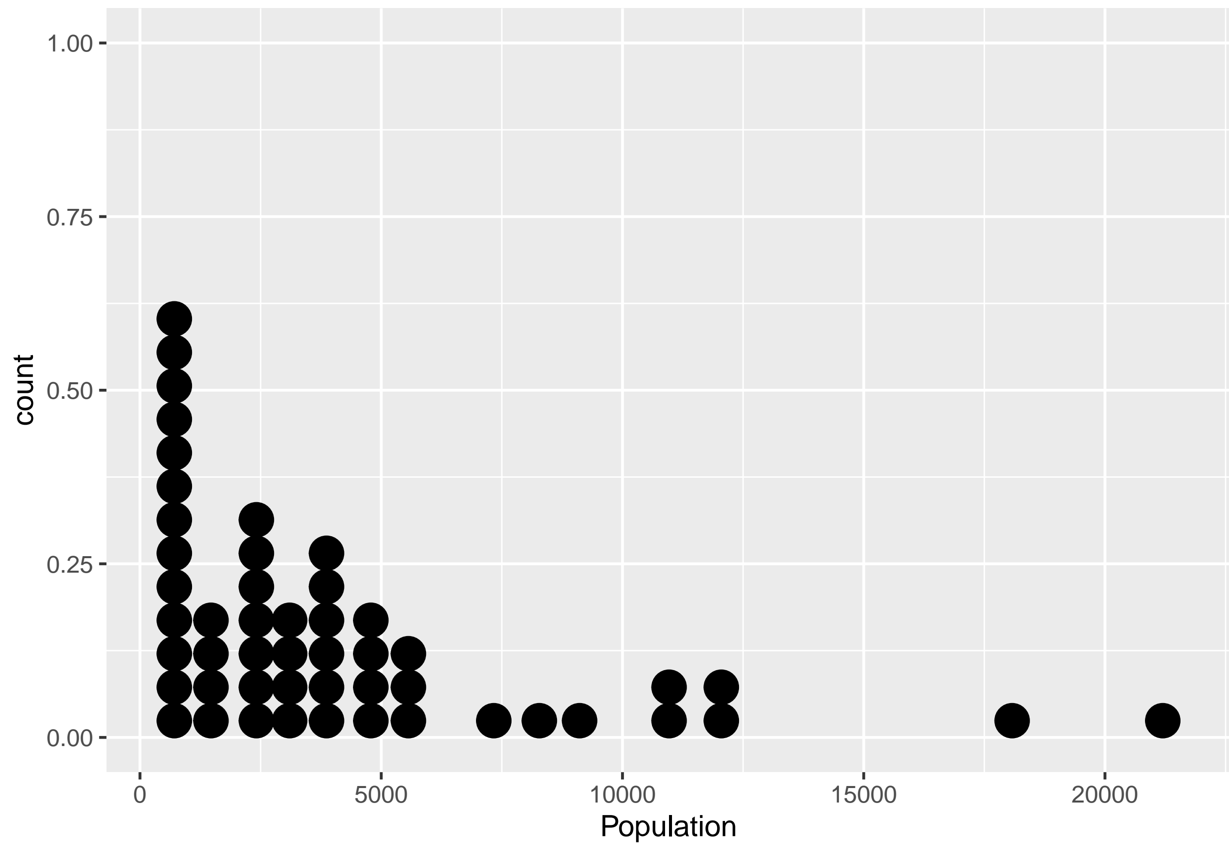



```
ggplot(state77,aes(Population)) + geom_histogram(bins=10)
```

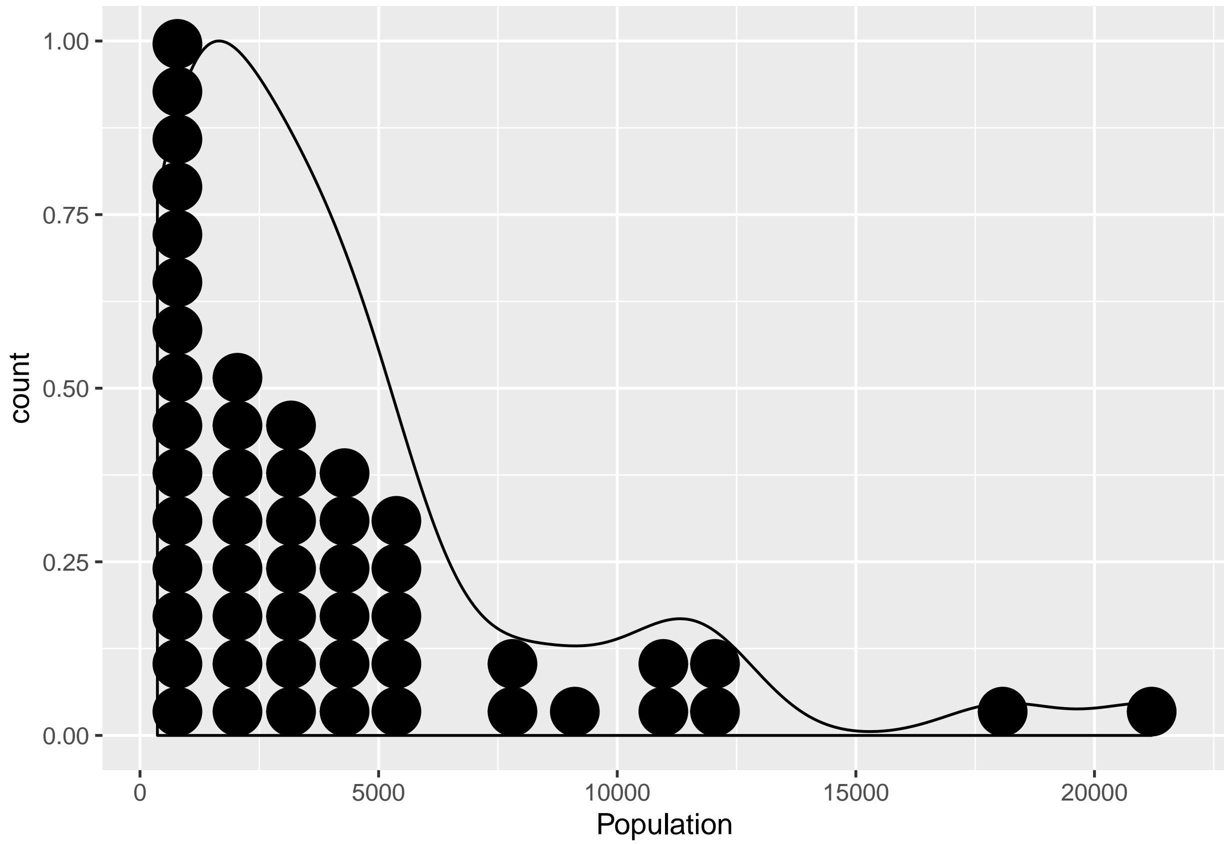



```r
ggplot(state77,aes(Population)) + geom_dotplot()
```

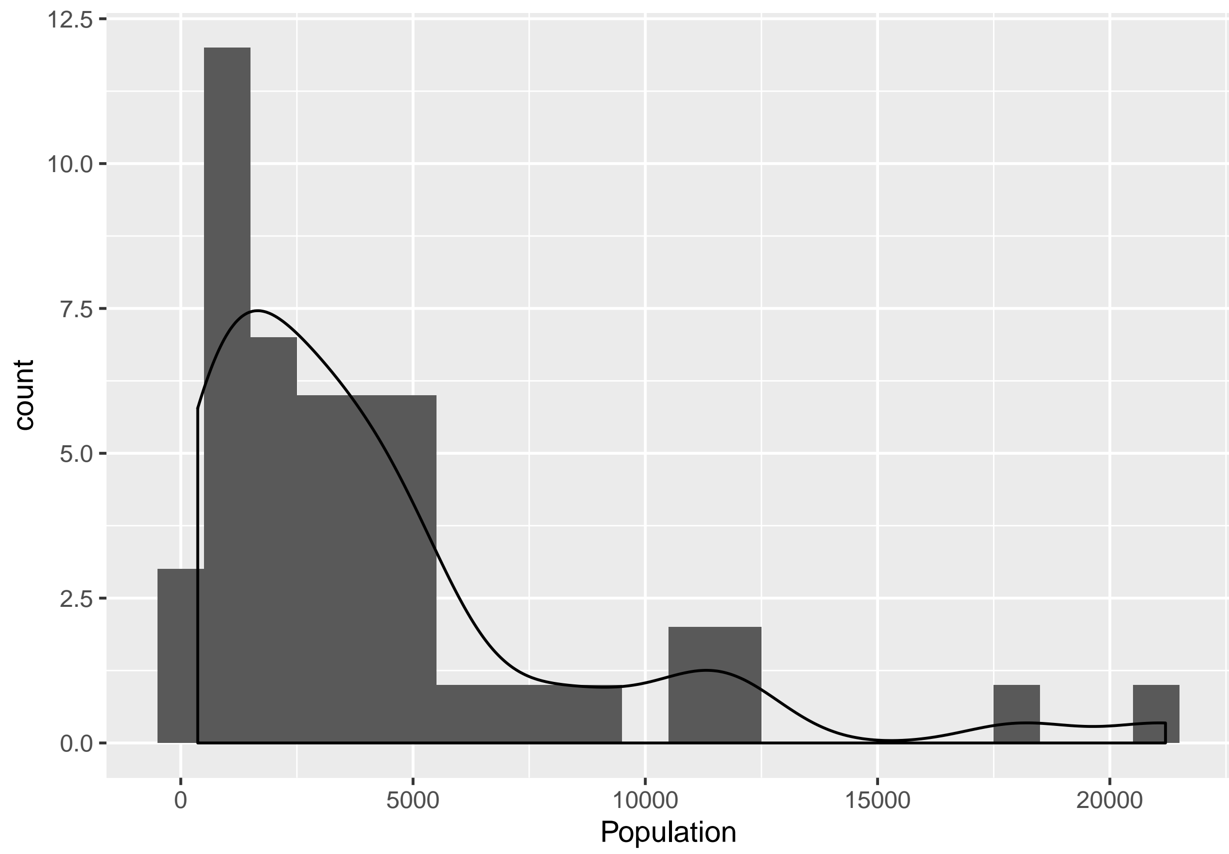
```
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```



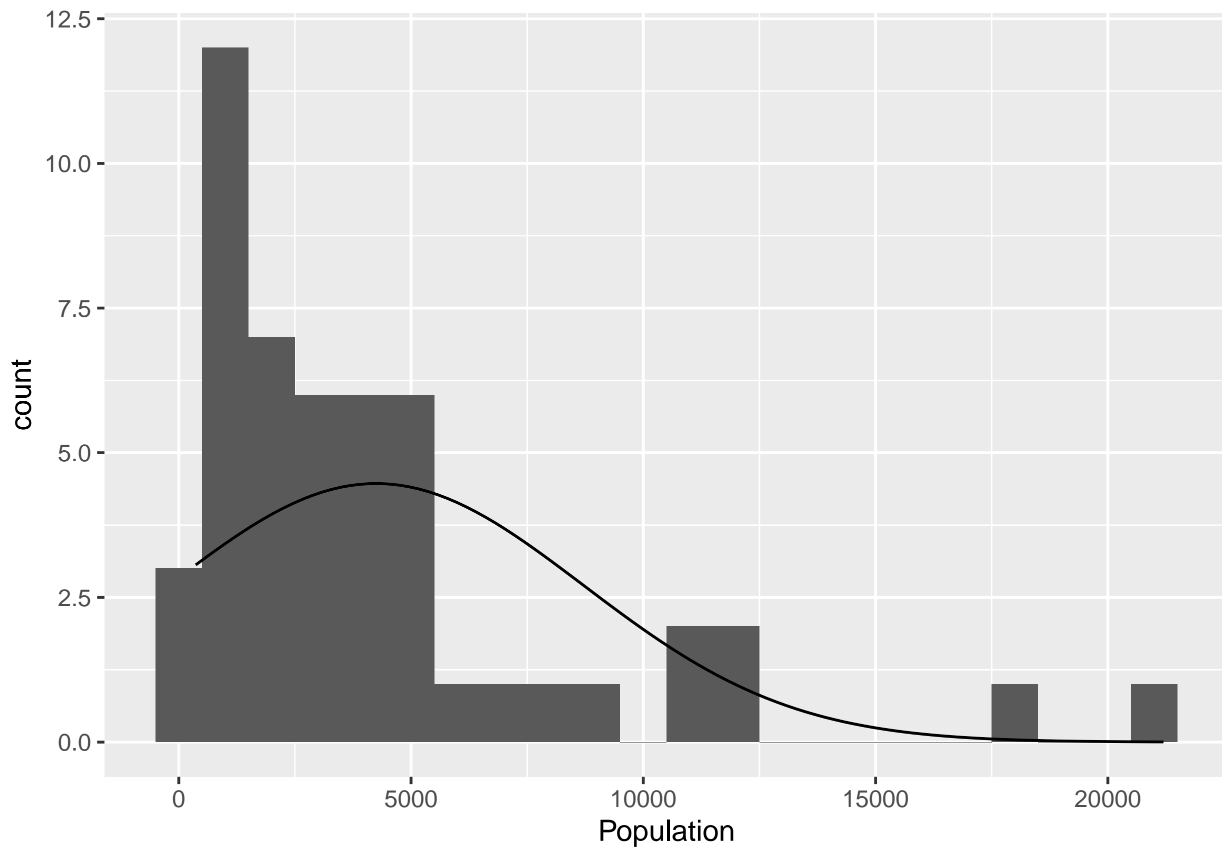

```
ggplot(state77,aes(Population)) +geom_dotplot(binwidth=1000) +geom_density(aes(y=..scaled..))
```



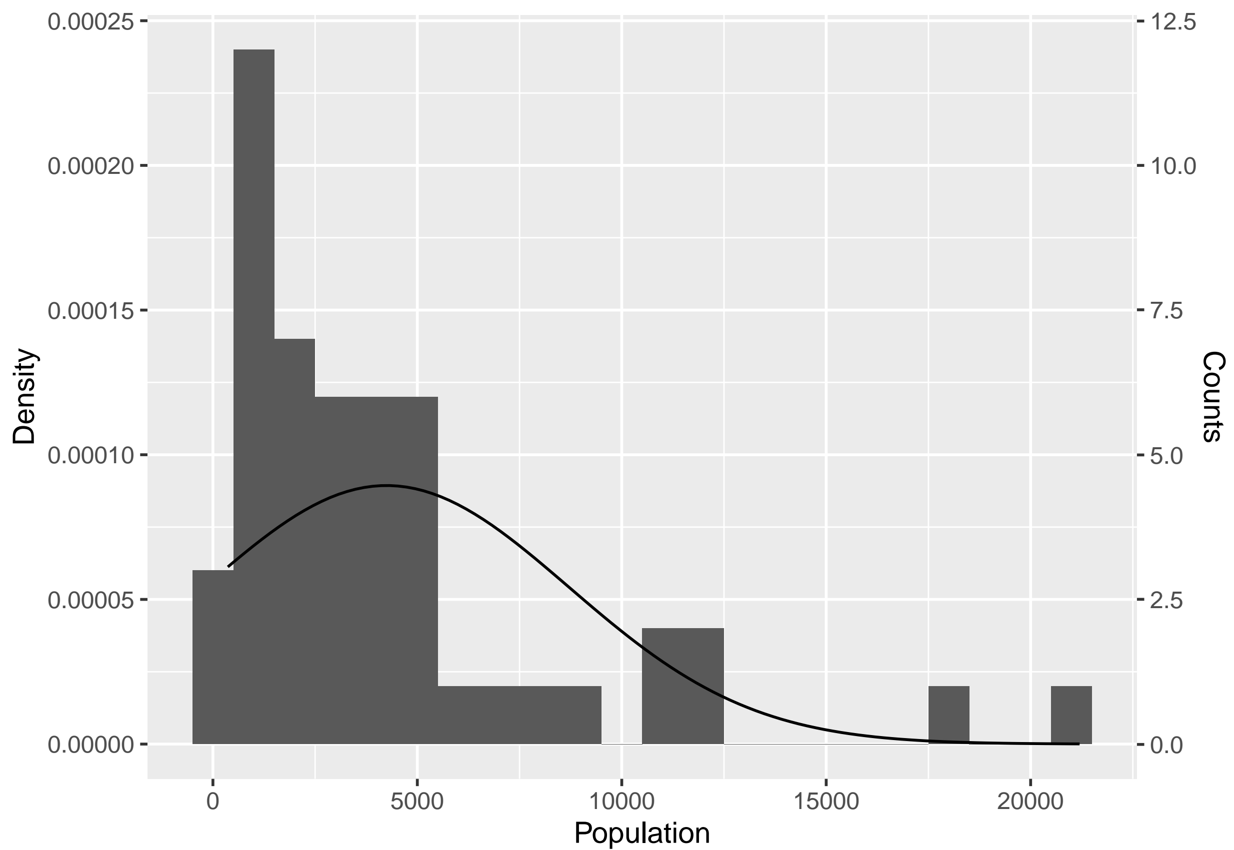

```
ggplot(state77,aes(Population)) +geom_histogram(binwidth=1000) +geom_density(aes(y=1000*..count..))
```




```
ggplot(state77,aes(Population)) +geom_histogram(binwidth=1000) +stat_function(fun= function(x) dnorm(x,
```



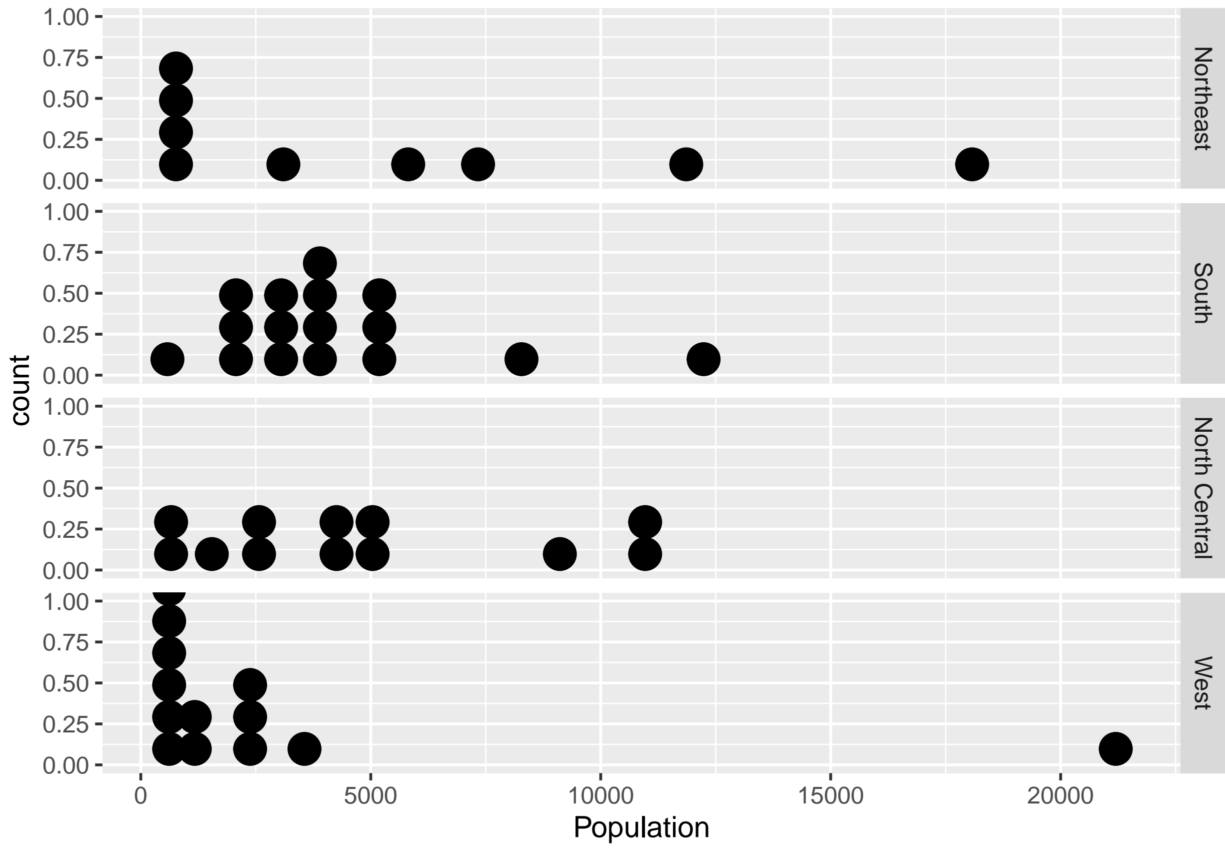

```
bw <- 1000
ggplot(state77,aes(Population)) + geom_histogram(aes(y=..density..),binwidth=bw) +
stat_function(fun=dnorm, args=c(mean=mean(state77$Population), sd=sd(state77$Population))) +
scale_y_continuous("Density",sec.axis=sec_axis(trans = ~ . * bw * nrow(state77), name = "Counts"))
```



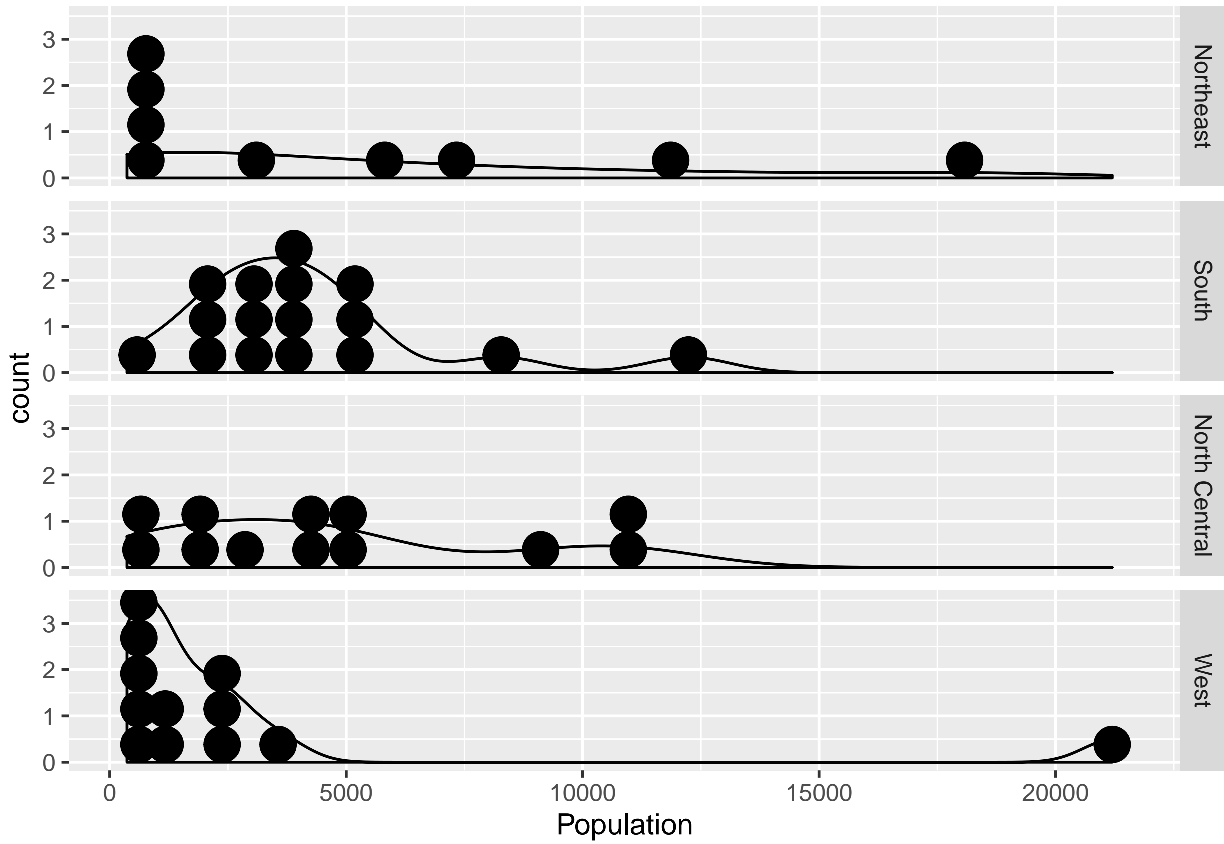
Panel Histograms by a Group

```
ggplot(state77,aes(Population)) + facet_grid(rows=vars(region)) + geom_dotplot()
```

```
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```

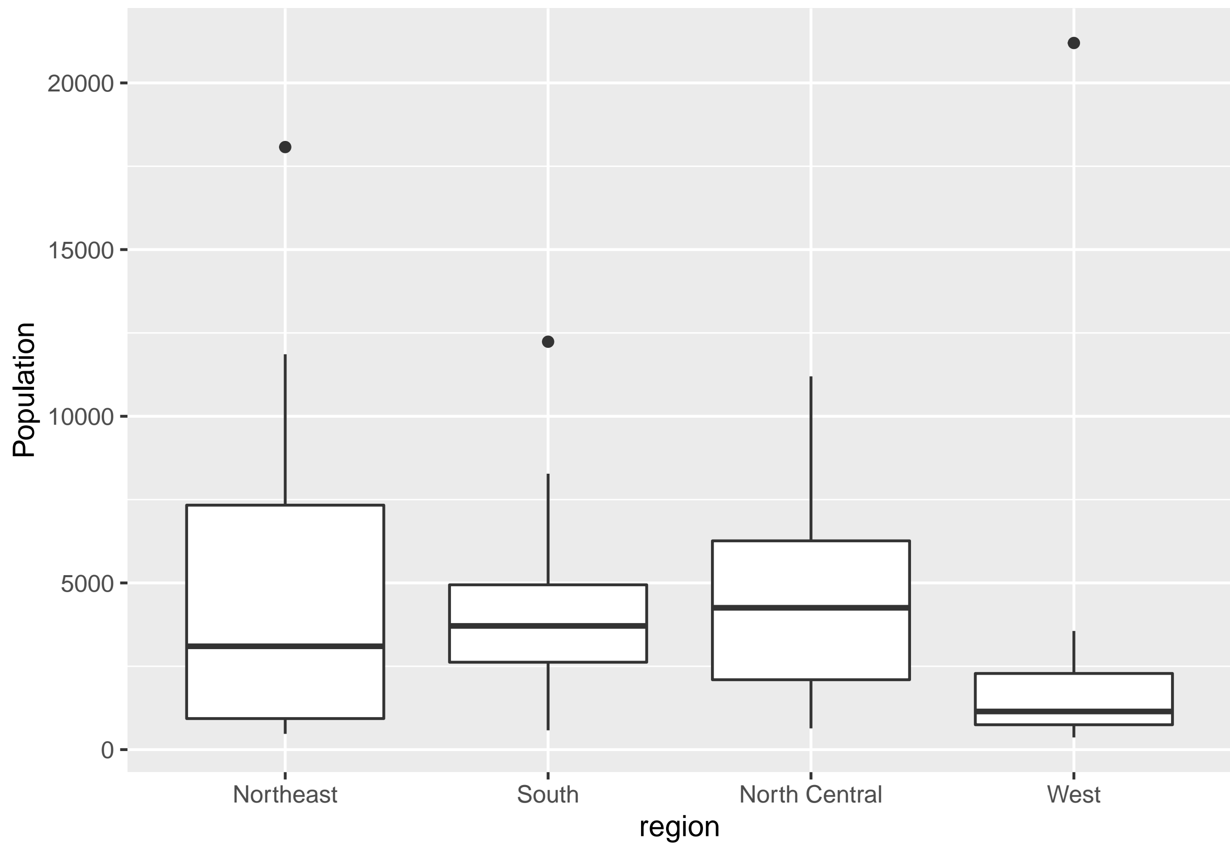



```
ggplot(state77,aes(Population)) + facet_grid(rows=vars(region)) + geom_dotplot(binwidth=750)+geom_densi
```

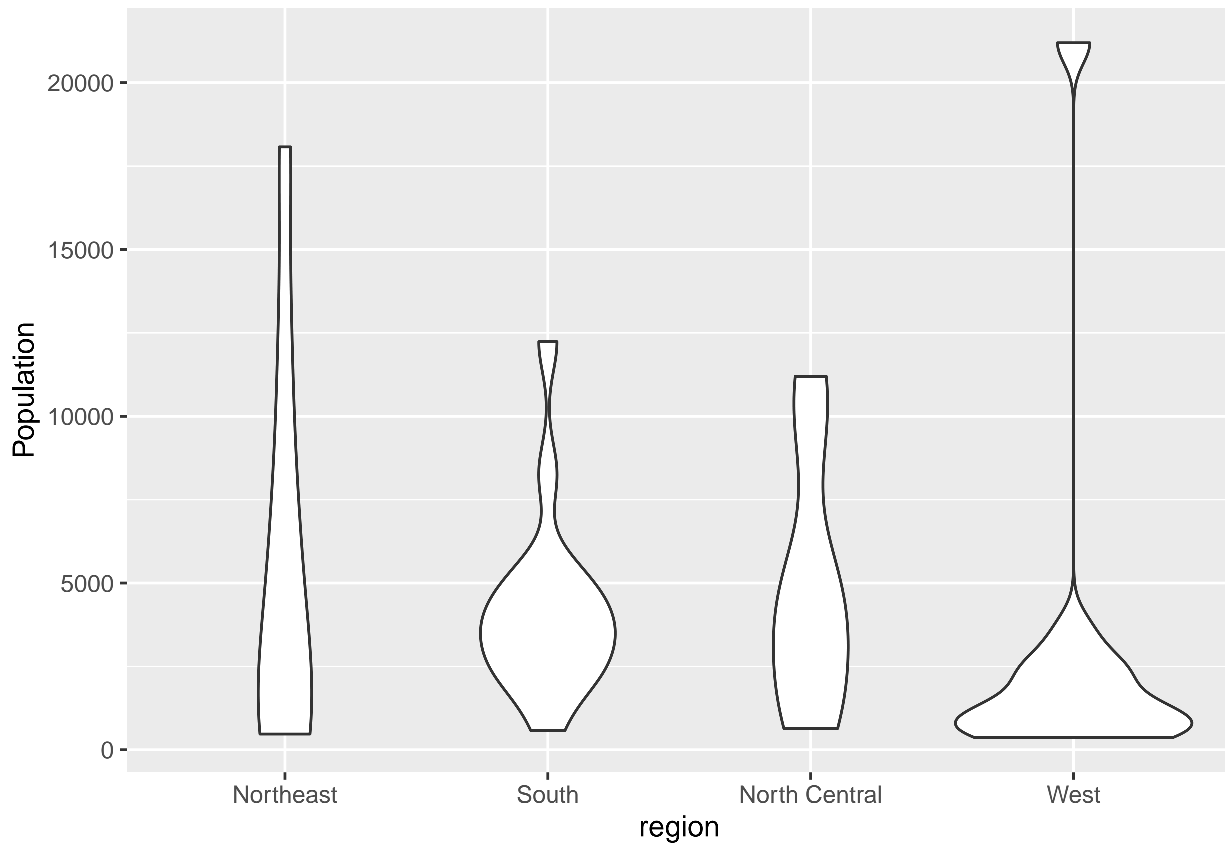


Making Boxplots

```
ggplot(state77,aes(x=region,y=Population)) + geom_boxplot()
```

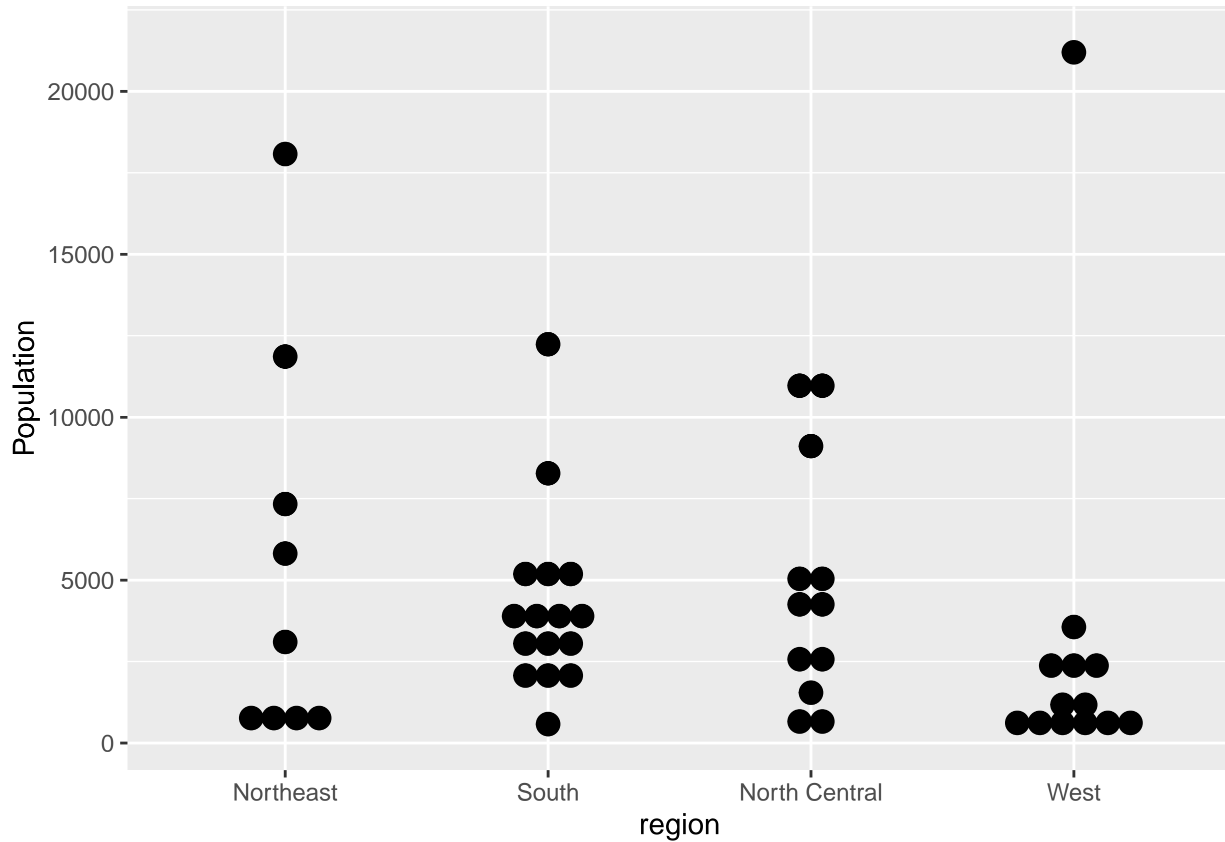



```
ggplot(state77,aes(x=region,y=Population)) + geom_violin()
```




```r
ggplot(state77,aes(region,Population)) + geom_dotplot(binaxis="y",stackdir="center")
```

```
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```



Saving Your Work

Saving Your Plots

```
ggsave("foo.png")
```

```
## Saving 6.5 x 4.5 in image
## `stat_bindot()` using `bins = 30`. Pick better value with `binwidth`.
```

Saving Your Tables

```
library(xtable)
print(xtable(state77 %>% group_by(region)%>% select(Population,Area) %>% summarize_all(list(mean=mean,s
```

```
## Adding missing grouping variables: `region`
```

result

Working in R Markdown

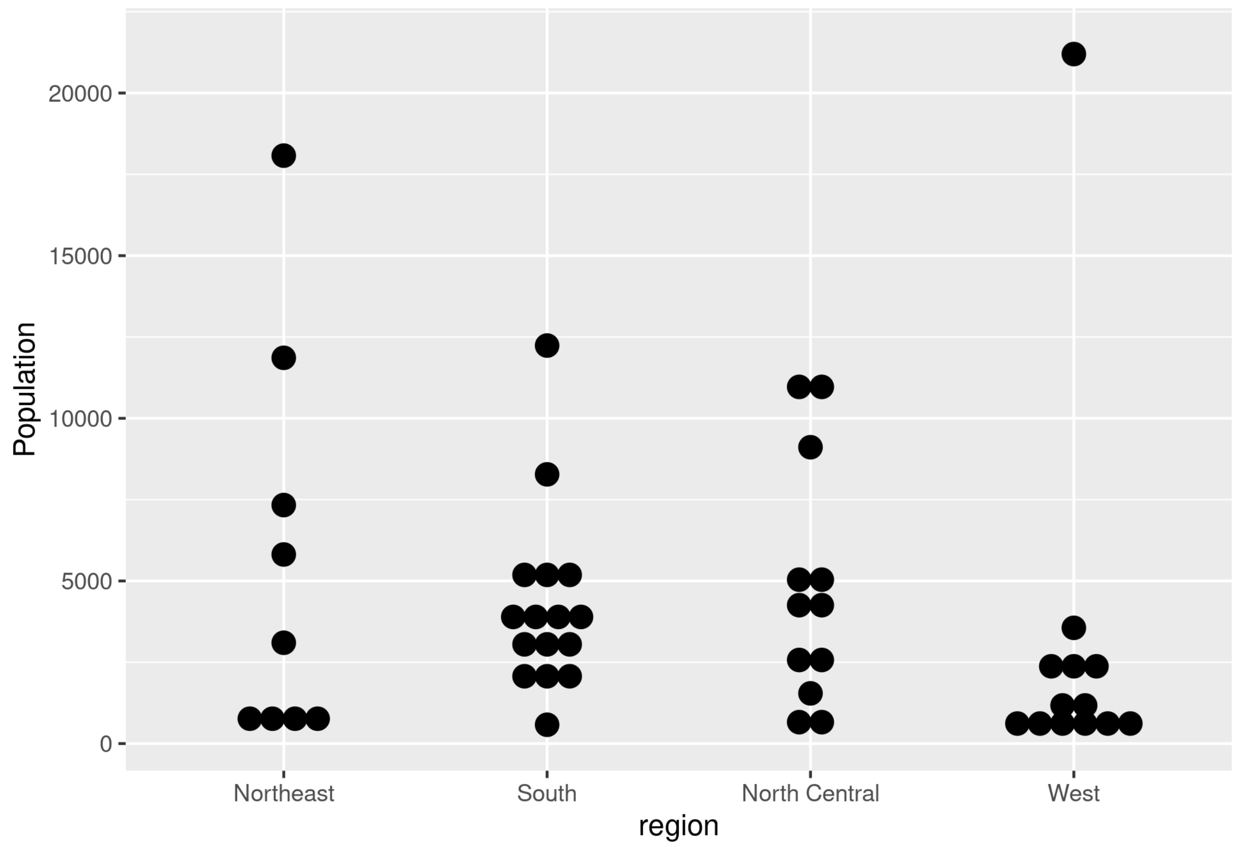


Figure 1: Just saved file.